Deep Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2024-08 Dozent: Karsten Flügge

Themen des Tages

Tag 4

Lernkurven
Überanpassung und Regularisierung
Hyperparameter-Optimierung
Stochastischer Gradienten Abstieg (SGD) Descent

Pause

Überanpassung (Overfitting)

- Definition: Überanpassung (**Overfitting**) tritt auf, wenn ein Modell zu genau auf die Trainingsdaten angepasst wird und somit auf neuen, unbekannten Daten **schlecht generalisiert**.

Übertraining ≈ Overfitting ≠ Underfitting

- **Symptome:**
  - Hohe Trainingsgenauigkeit, aber niedrige Test/Validierungsgenauigkeit.
  - Modell versagt in der Praxis

Überanpassung (Overfitting)

- Historische **Ursachen** von Überanpassung

* **Zu viele Parameter** im Modell, hohe Kapazität (z.B. tiefe neuronale Netze)
* Zu viele Trainingsdaten mit **hoher Variabilität**
* Zu wenig Trainingsdaten: häufig dass das Modell die Trainingsdaten "**auswendig lernt**" anstatt zu generalisieren.
* **Falsche Daten** oder fehlerhafter Datenaufbereitung
* **Bug** im Neuronalen Netz

* **Fehlende Regularisierung … !**

Regularisierung gegen Überanpassung

- **Vermeidung** von Überanpassung

Glücklicherweise wurden in den letzen Jahren viele Methoden entwickelt, um das alte Problem von Überanpassung sehr gut in den Griff zu bekommen:

A) Im Model

- **Skip-Verbindungen**
  Residual Connections: Weiterschleifen von unverfälschten Signalen in tiefere Schichten (Tag 7)
- **Batch Normalization**
- Ausblick: Abziehen von Schwerpunkt und teilen durch Varianz, pro batch(!) auch in tieferen Schichten
- **Dropout:** Zufälliges “Ausschalten” von Neuronen während des Trainings, um das Modell robuster zu machen.
  statt das alle Information in einem Neuron konzentrier werden kann, forciert dieser Ansatz 'holographisches' lernen
- **L1- und L2-Regularisierung:** Hinzufügen eines Penalisierungsterms zu der Verlustfunktion, um große Gewichte zu verhindern.
  allgemeiner kann man die Fehlerfunktion mit eigenen Kriterien erweitern (auch als Schicht!)

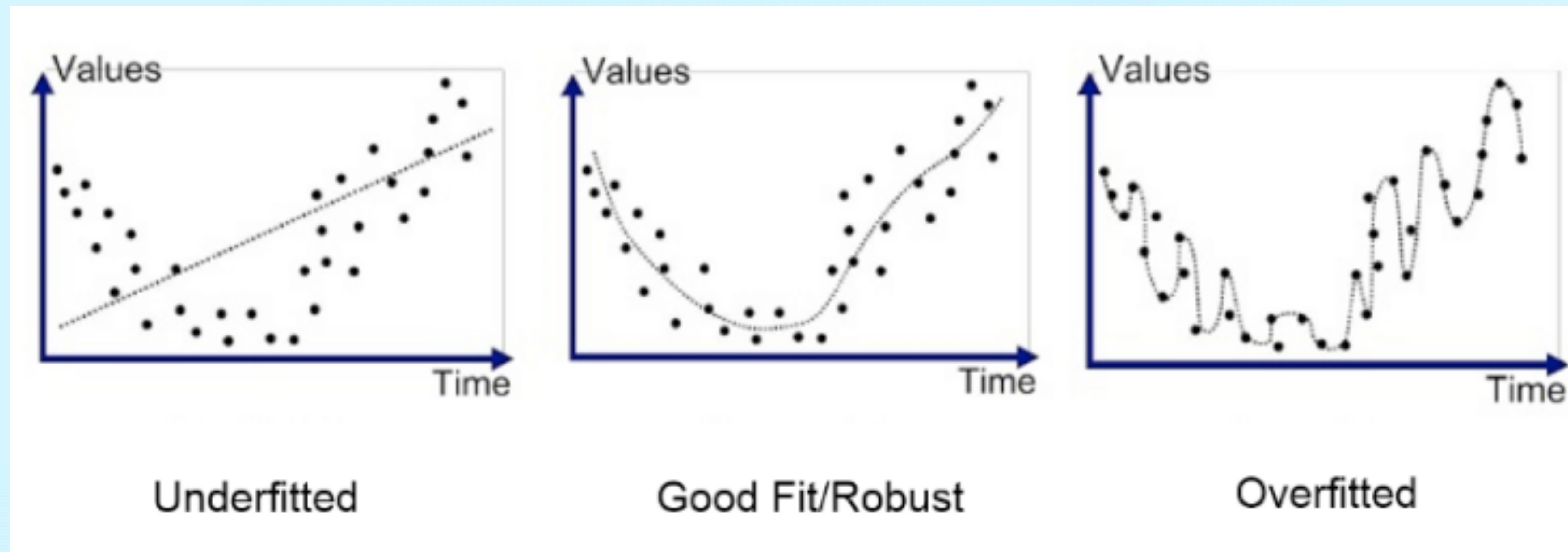
B) Im Fehler

- **L1- und L2-Regularisierung**

C) Daten:

- **Datenbereiningung und Vorbereitung**
- **Synthetische Daten &**
- **Datenaugmentation:** Künstliche Erhöhung der Trainingsdatenmenge durch Transformationen (z.B. Rotationen, Skalierungen).
  sehr anschaulich bei OCR (Texterkennung) : man kann die Texte auf verschiedenste Weisen '**stören**' und das Netz lernen lassen

Überanpassung Beispiele




```
 Stellen wir uns vor, dass wir versuchen, eine Funktion y = f(x) zu lernen,
die von einem unbekannten x-Wert auf einen
# entsprechenden y-Wert schätzen soll. Wir haben nur zwei Trainingsdatenpunkte:
#
# * (x1, y1) = (4, 17)
# * (x2, y2) = (9, 74)
#
# Wir möchten eine lineare Funktion in der Form y = ax + b erlernen,
die diese beiden Punkte am besten passt.
```

* **Zu viele Parameter** im Modell, hohe Kapazität (z.B. tiefe neuronale Netze)

```
# **Überanpassung durch Lernen einer zu komplexen Funktion:**
#
# Wenn wir eine komplexe Funktion wie y = a(x^5) + b(x^3) + c(x) + d lernen, wird sie sich sehr schnell an die beiden
# Trainingsdatenpunkte anpassen (d.h., sie wird sehr genau diese Punkte treffen). Dies ist jedoch kein guter Modell, da es
# wahrscheinlich nicht auf neue Daten übertragbar ist und daher nur zu wenigen Datapunkten passt.
```

* Zu viele Trainingsdaten mit **hoher Variabilität**

* **Zu wenig Trainingsdaten**: häufig dass das Modell die Trainingsdaten “**auswendig lernt**” anstatt zu generalisieren.
* Oder das Muster wird erst bei mehreren Punkten erkennbar

* **Falsche Daten** oder fehlerhafter Datenaufbereitung
* **Bug** im Neuronalen Netz

```
# **Überanpassung durch Lernen einer Funktion mit einem zu großen Koeffizienten:**
#
# Wenn wir eine lineare Funktion wie y = 1000x + b lernen, wird sie sich sehr schnell an die beiden Trainingsdatenpunkte anpassen.
# Dies ist jedoch kein guter Modell, da es wahrscheinlich nicht auf neue Daten übertragbar ist und daher nur zu wenigen Datapunkten
# passt.
```

* Fehlende Regularisierung …

Datenaugmentation

Achtung, man kann auch übertreiben:

Modifizieren: nehme existierende Trainingsdaten und modifiziere sie (rotieren, skalieren, Farben ändern, Helligkeit ändern, **Rauschen**)

Synthetische Daten: erzeuge ganz eigene Trainingsdaten

Wenn zu viele synthetische Daten erzeugt werden, können die originalen Daten untergehen. Gegenteil des Regularisierungseffektes welchen wir gewünscht haben.

Dropout

Einfache Idee: Schalte in einer Zwischenschicht zufällig Aktivierungen aus (auf 0)

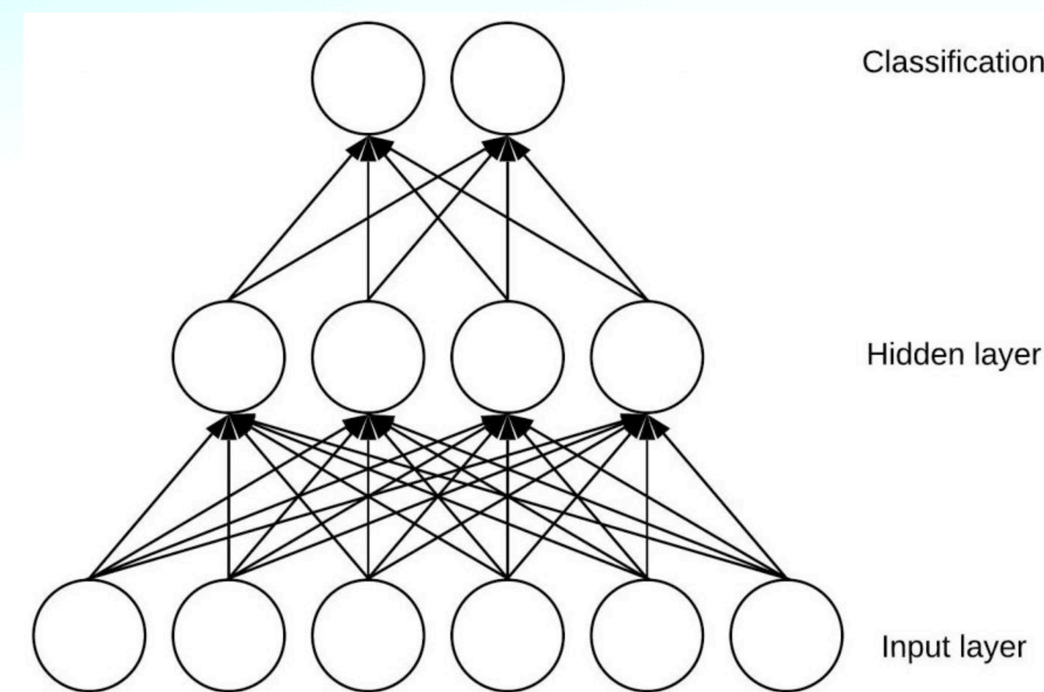
Ohne Dropout: Grossmutter Neuron: Ein einziges Perceptron kann für ein Trainingspunkt verantwortlich sein.

Mit Dropout:
Holographisches Lernen: Verteile alle Informationen auf alle Gewichte.

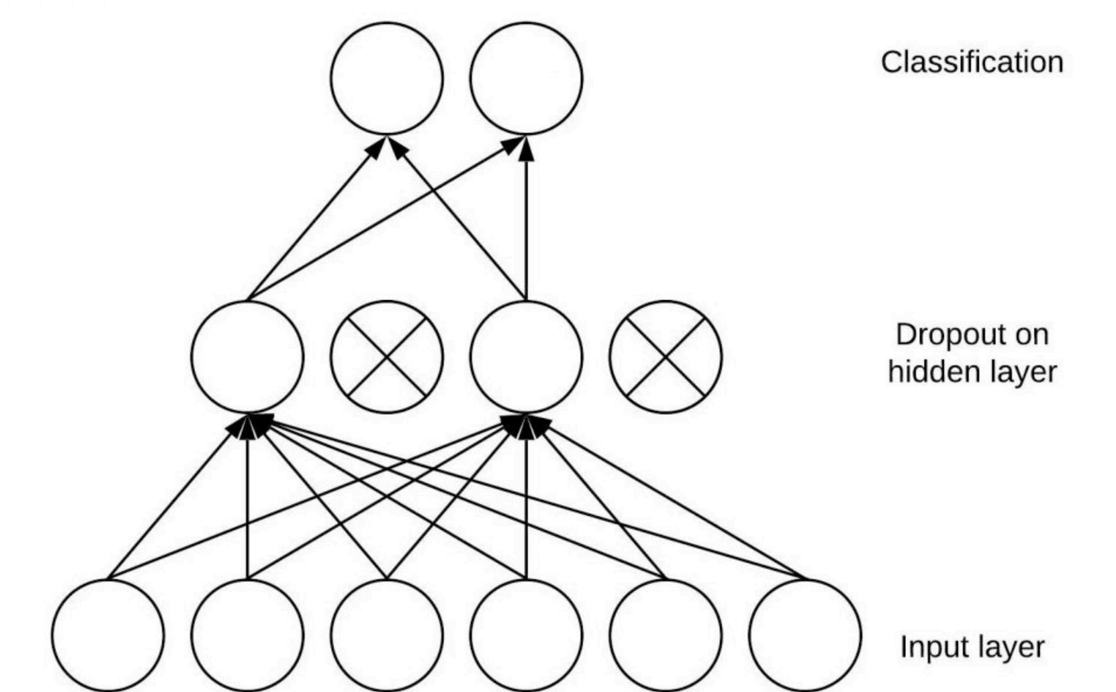
```
dropout_rate = 0.1 # … 0.5 # 0.2 : 20% dropout werden ignoriert / auf 0 gesetzt
model = torch.nn.Sequential(
    torch.nn.Linear( 28 * 28 , 10, bias=True), # most trivial
    torch.nn.Dropout(dropout_rate) # dropout layer
)
```

Typische Werte 0.1 bis 0.5

⚠️ Achtung!
Vor dem Evaluieren/
Testen muss dropout
kurz auf 0 gesetzt werden!
via model.eval() in pytorch



Without Dropout



With Dropout

Pytorch train/eval

⚠️ Achtung!

Vor dem Evaluieren/Testen muss dropout in **pytorch model.eval()** aufgerufen werden

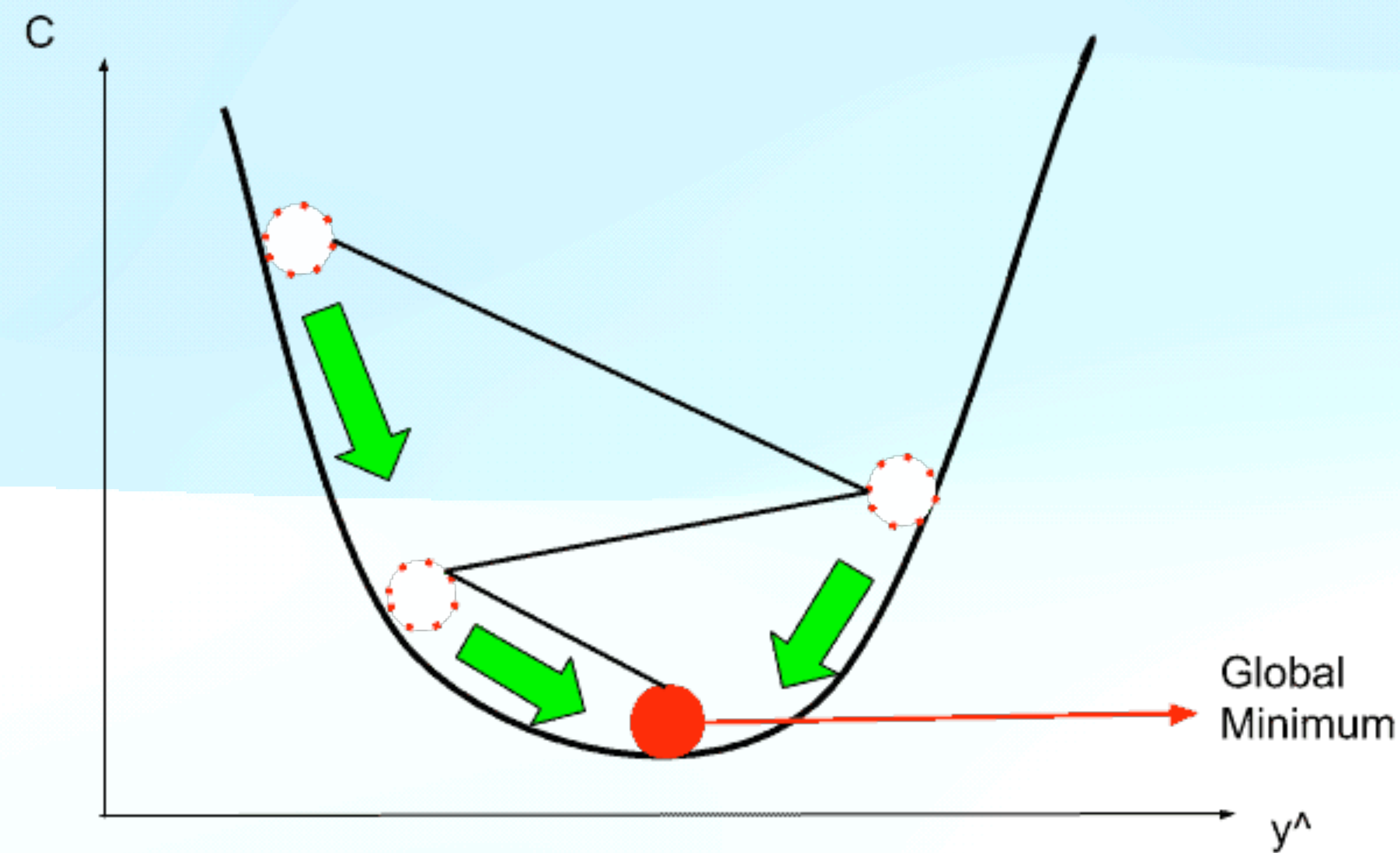
z.B. damit bei accuracy dropout auf 0 gesetzt wird

Vor dem Training dann (wieder) **model.train()**

SGD Stochastic Gradient Descent

Stochastischer Gradienten Abstieg

Optimierung in Richtung der größten Änderung / des größten Fehlers



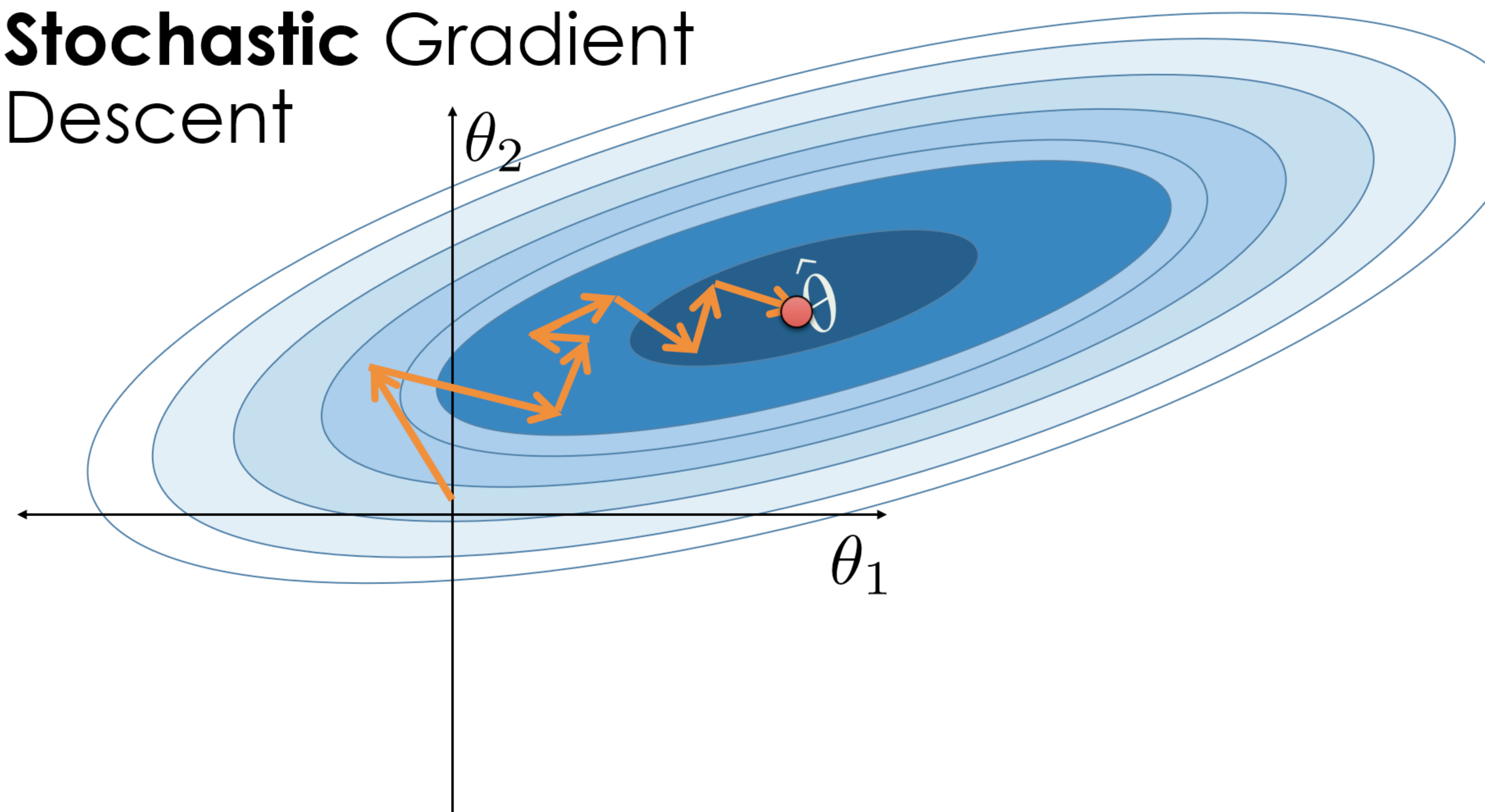
SGD Stochastic Gradient Descent

Stochastischer Gradienten Abstieg

Optimierung in Richtung der größten Änderung / des größten Fehlers

Iterative Methode, um effizient das **Lernkriterium** zu optimieren.
Fehler (Error, Loss) minimieren, Genauigkeit/accuracy maximieren …

Warum "stochastisch"?
Weil wir aus dem oft riesigen Satz von Trainingsdaten in jedem Schritt **zufällig** welche auswählen. Zudem wird oft nicht der echte **Gradient** (Ableitung) der Lernfunktion benutzt, sondern eine **Approximation.**



SGD Stochastic Gradient Descent

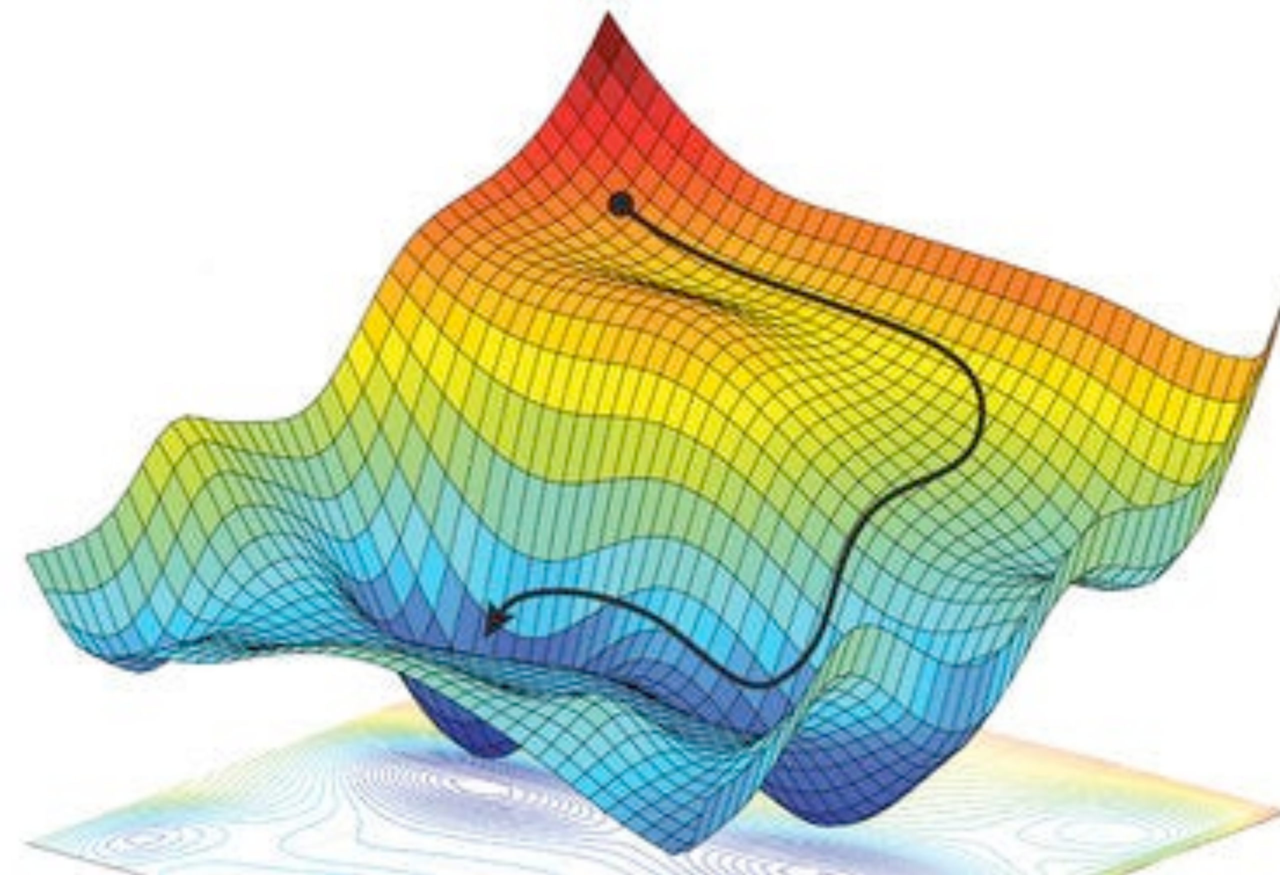
Stochastischer **Gradienten** Abstieg

Optimierung in Richtung der größten **Änderung** / des größten Fehlers

Ein **Gradient** einer Funktion ist die **Ableitung** also
die **Änderungsrate** von der Funktion an einem Punkt.

In einer Dimension gibt das Vorzeichen an, ob die Funktion hier wächst oder fällt.
In mehreren Dimensionen ergeben mehrere Vorzeichen zusammen die **Richtung** der stärksten Änderung.

Man nimmt einfach die normale Ableitung von jeder Komponente und erhält so den **Vektor.**



$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

Non-Stochastic Gradient Descent

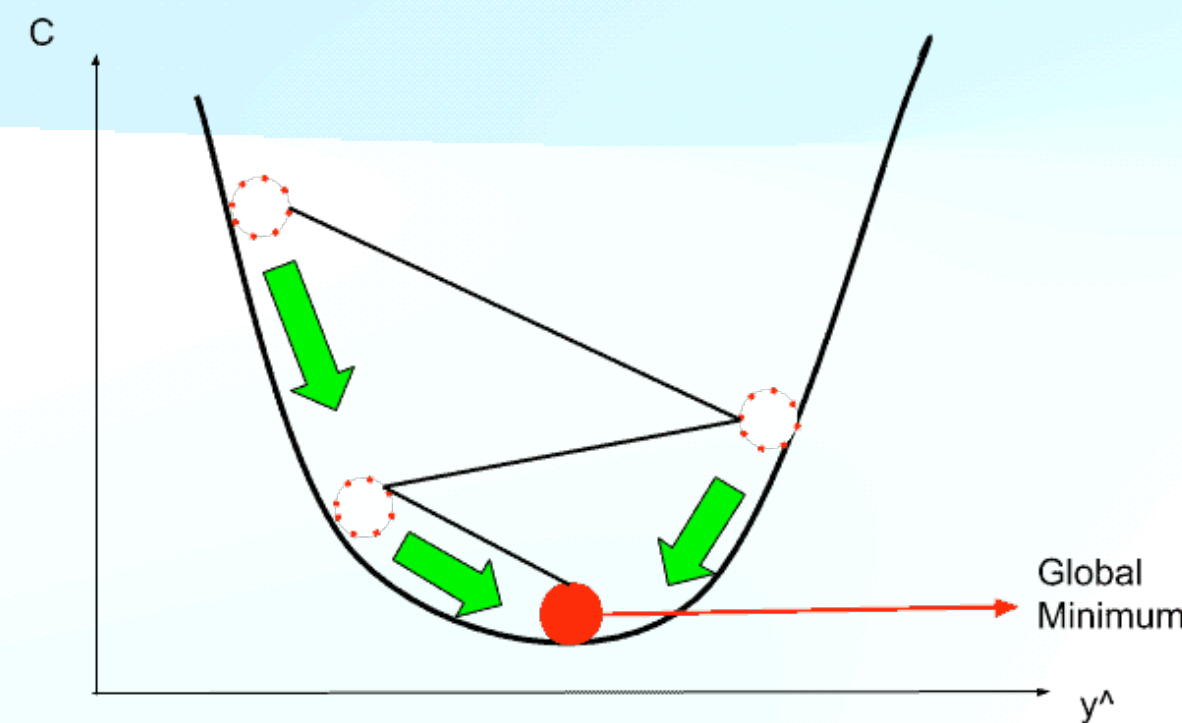
Einen deterministischen Gradienten Abstieg habt ihr gegebenenfalls schon in der Schule gesehen:

Mit dem **Newton Verfahren** kann man recht schnell minima auffinden.

Man kann aber auch andere **primitivere Verfahren** anwenden,

z.B. gehe 1 Schritt nach rechts, wenn der Wert kleiner wird gehe noch einen Schritt, ansonsten gehe mit halber Schrittweite in andere Richtung...

Dies wäre ein Beispiel eines **gradienten-freien Verfahrens**



💡 Gerade wenn man vor Gradienten noch Angst hat, kann es hilfreich sein, ein primitiveres Verfahren selbst umzusetzen.

<u>Non</u>-Gradient Descent

Gradienten-freie Verfahren

Bis jetzt wurde kein Verfahren gefunden welches schneller lernt als Variationen vom Gradienten Abstieg. dennoch gibt es gute Gründe die Augen nach Alternativen offen zu halten:

1. Das menschliche **Gehirn** scheint fähig zu sein gut zu lernen **ohne implizit oder explizit Gradienten** zu berechnen!

2. Gradienten lassen sich schwer in **analogen photonen neuronalen Netzen** berechnen

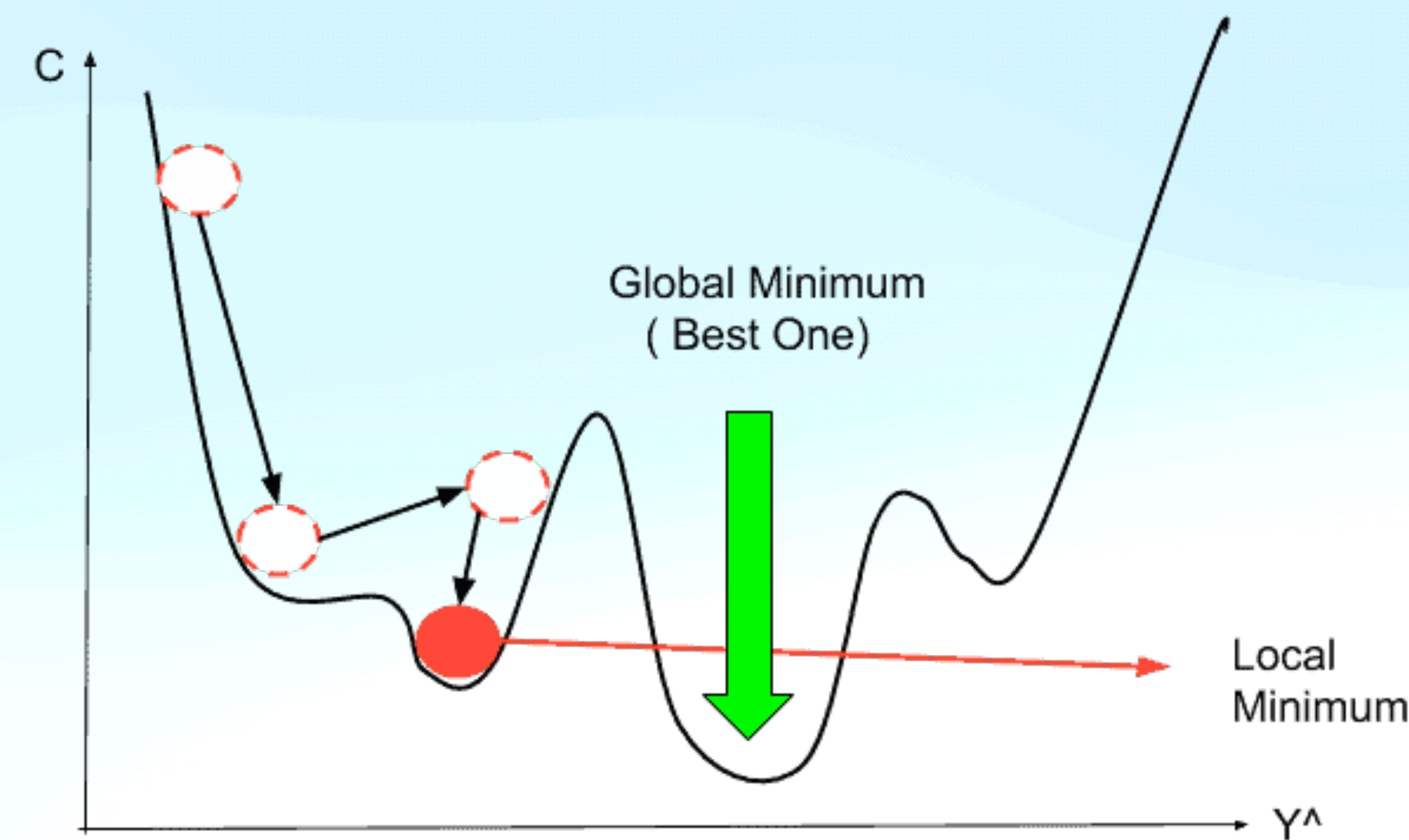
3. Außerdem primitiven Ansatz den Wert des nächsten Schrittes zu raten, gibt es noch andere Ansätze:

4. Im **Forward Forward** network war es zunächst ein Wunder wie das Netzwerk überhaupt etwas lernen kann. (Video / paper highly recommended!)

SGD Lokale Minima

Theoretisch ist kein Verfahren frei von der Gefahr, am globalen Minimum vorbei zu suchen und stattdessen nur lokale Minima zu finden.

Als kleines selbst konstruiertes Beispiel nehmen man einfach die Sinusfunktion welche ans Stelle 10000000 einmal auf -2 heruntergeht und sonst zwischen +1 und -1 herum wabert.



No-Free-Lunch-Theorem: es gibt keine allgemein gültige Optimierungsstrategie, die für alle Probleme gleich gut funktioniert.

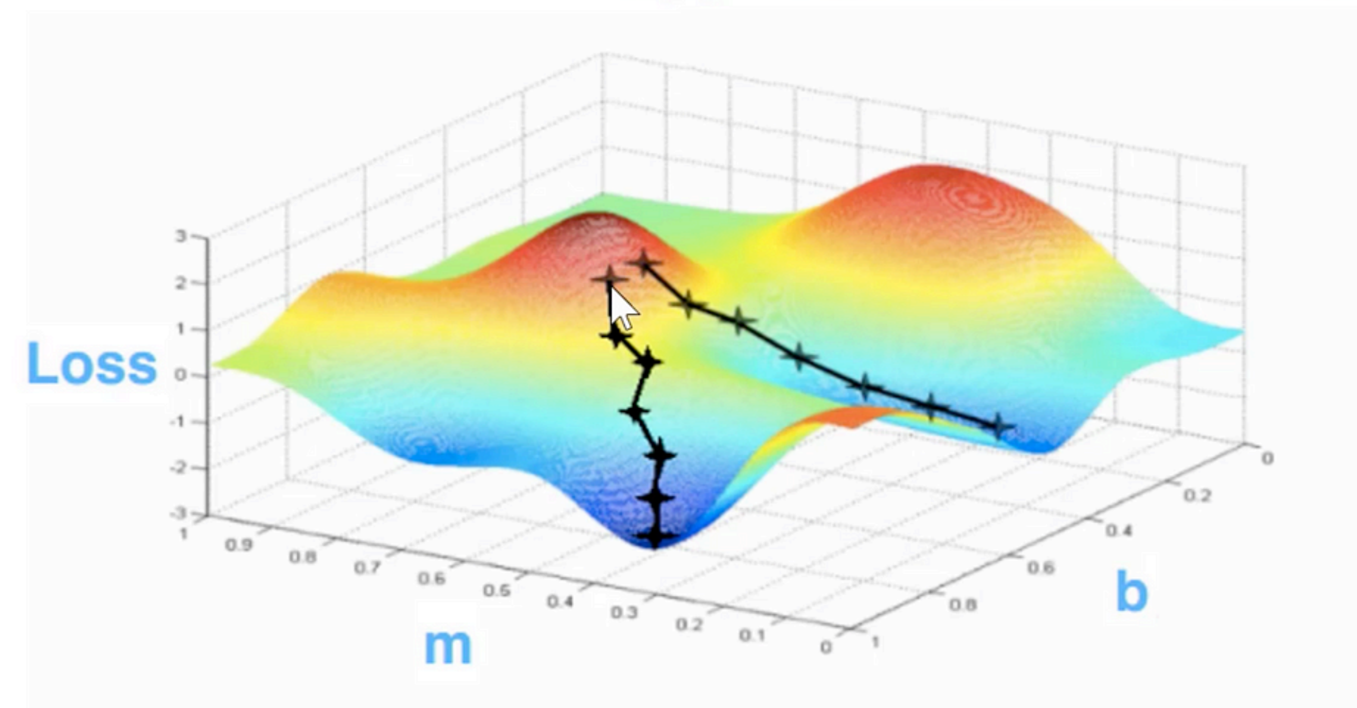
Praktisch ist die Gefahr in einem lokalen Minimum stecken zu bleiben bei neuronalen Netzen aber fast irrelevant:

Hohe Dimensionen

Lemma XYZ: "Je grösser die Dimension der Trainingsdaten, desto unwahrscheinlicher, dass man in einem lokalen Minimum stecken bleibt."

Da man beim Stochastischen Gradienten Abstieg zudem stets eine andere zufällige Auswahl aus den Trainingsbeispielen selektiert, gibt es quasi immer etwas zu lernen.

⚠️ sukzessive Optimierung für verschiedene Trainingsdaten idealer Weise ohne den Lernfortschritt vorheriger Trainingsdaten zu nihilieren. Dafür gibt es u.a. den Hyperparameter **learning-rate**, mit welcher man steuern kann, wie weit ein Schritt in Richtung des Gradienten sein soll.



Gradient vom trivialen Netzwerk

```
def gradient(input, weights):
  output = aktivierung(input, weights)
  # Ableitung der Fehlerfunktion machen jetzt
  fehler_ableitung = -(target - output)
  grad = np.array([1, input[0], input[1]]) * fehler_ableitung
  return grad
```

```
f = fehler(input, weights)= 0.5 * (target - output) ** 2  # output y hängt von Gewichten ab
f = fehler(i, w)= 0.5 * (t - y(w)) ** 2

df / dw = f'(y(w)) * y'(w)  # Kettenregel

f' =  0.5 (t^2 -2ty + y^2)' =  0.5*-2*t + 0.5*2*y  = -t + y =  -(target - output)
```

*y = w0 + w1*x1 + w2*x2*

```
y' = dy/dw = [dy/dw0, dy/dw1, dy/dw2] = [1, x1, x2] = [1, input[0], input[1]]
```

Hyperparameter

Hyperparameter

Hyperparameter sind Parameter eines Modells, die **nicht** während des Trainings **gelernt** werden, sondern **vorher festgelegt** werden müssen.

Beispiele hierfür sind die **Lernrate**, die **Anzahl der Layer** in einem neuronalen Netz, die **Batch-Größe** oder der **Regularisierungsterm.** Man kann aber beliebig viele Stellschrauben bei der Initialisierung oder beim Training mitgeben.

Hyperparameter-Optimierung

Hyperparameter Optimierung: Suchen der besten Werte für diese Parameter gefunden werden, um die Modellleistung zu maximieren.

Statt die Hyperparameter manuell auszuprobieren kann man sie auch programmatisch durch testen:

Entweder indem man sie **sukzessive** durchgeht:

```
# Hyperparameter-Raum definieren
hyperparameter_space = {
    'learning_rate': [0.001, 0.01, 0.1],
    'batch_size': [32, 64, 128],
    'dropout':[0,0.1,0.3,0.5]

    'error_function': [MSE, ABS, L1, Custom], ...
    'optimizer': [optim.SGD, optim.Adam]
}
```

Oder indem man sie **zufällig** auswählt aus einem Intervall

Hyperparameter-Optimierung MNIST

Beispiel MNIST mit fixed wall time:
Welche Konfiguration führt **in begrenzter Zeit** zum besten Ergebnis?

Bei begrenzter Epochenzahl.

```
# Hyperparameter-Raum definieren
hyperparameter_space = {
    'learning_rate': [0.001, 0.01, 0.1],
    'batch_size': [32, 64, 128],
    'dropout':[0,0.1,0.3,0.5]

    'error_function': [MSE, ABS, L1, Custom], ...
    'optimizer': [optim.SGD, optim.Adam]
}

for lr in hyperparameter_space['learning_rate']:
 for bs in hyperparameter_space['batch_size']:
  for drop in hyperparameter_space['dropout']:
```

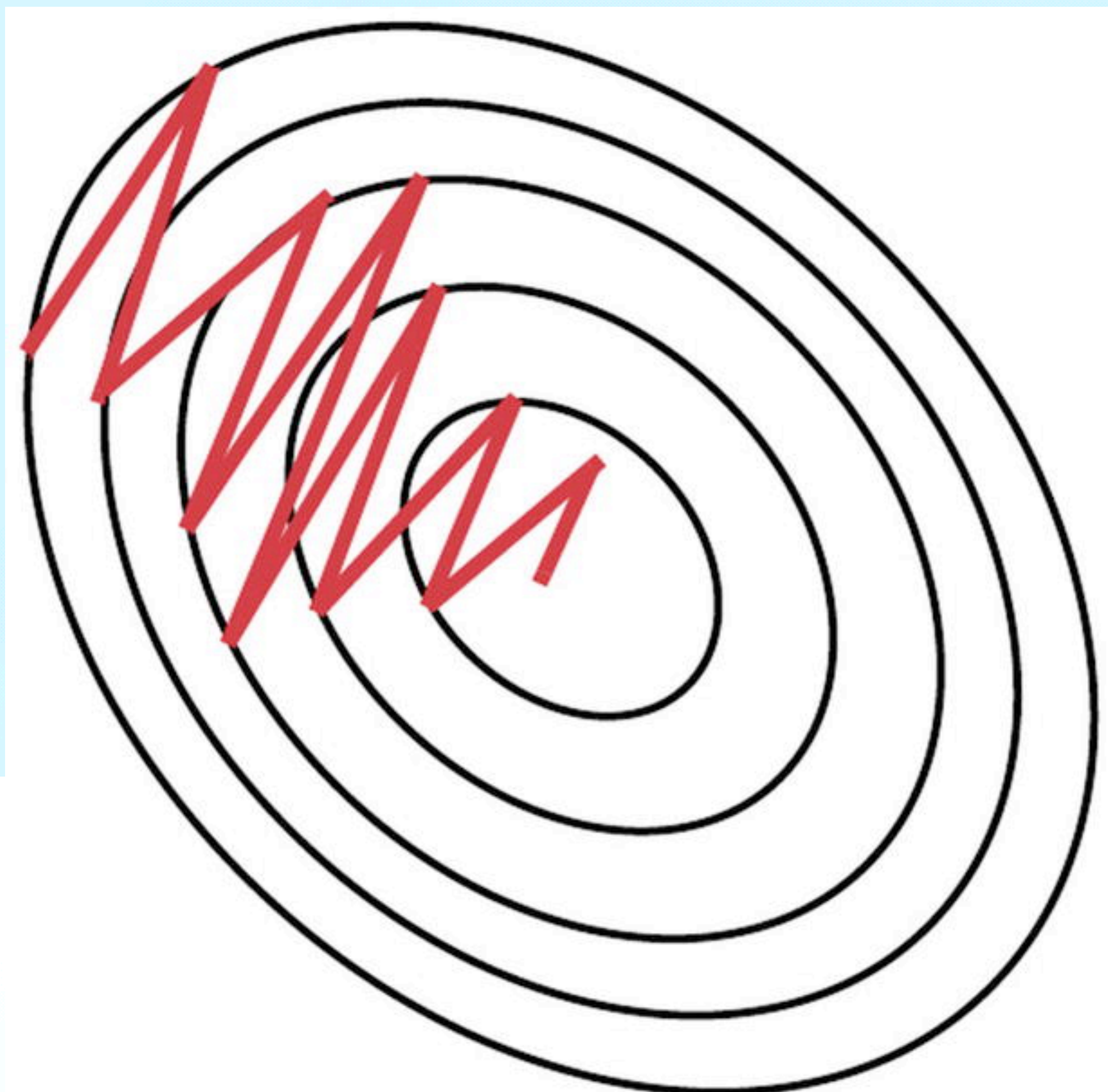
Lernrate

Mit dem Hyperparameter "**Lernrate**" kann man Steuern wie schnell oder aggressiv das Netz neue Informationen absorbiert. Konkret bedeutet das wie stark der Gradient beim Update die vorherigen Gewichte beeinflussen soll.

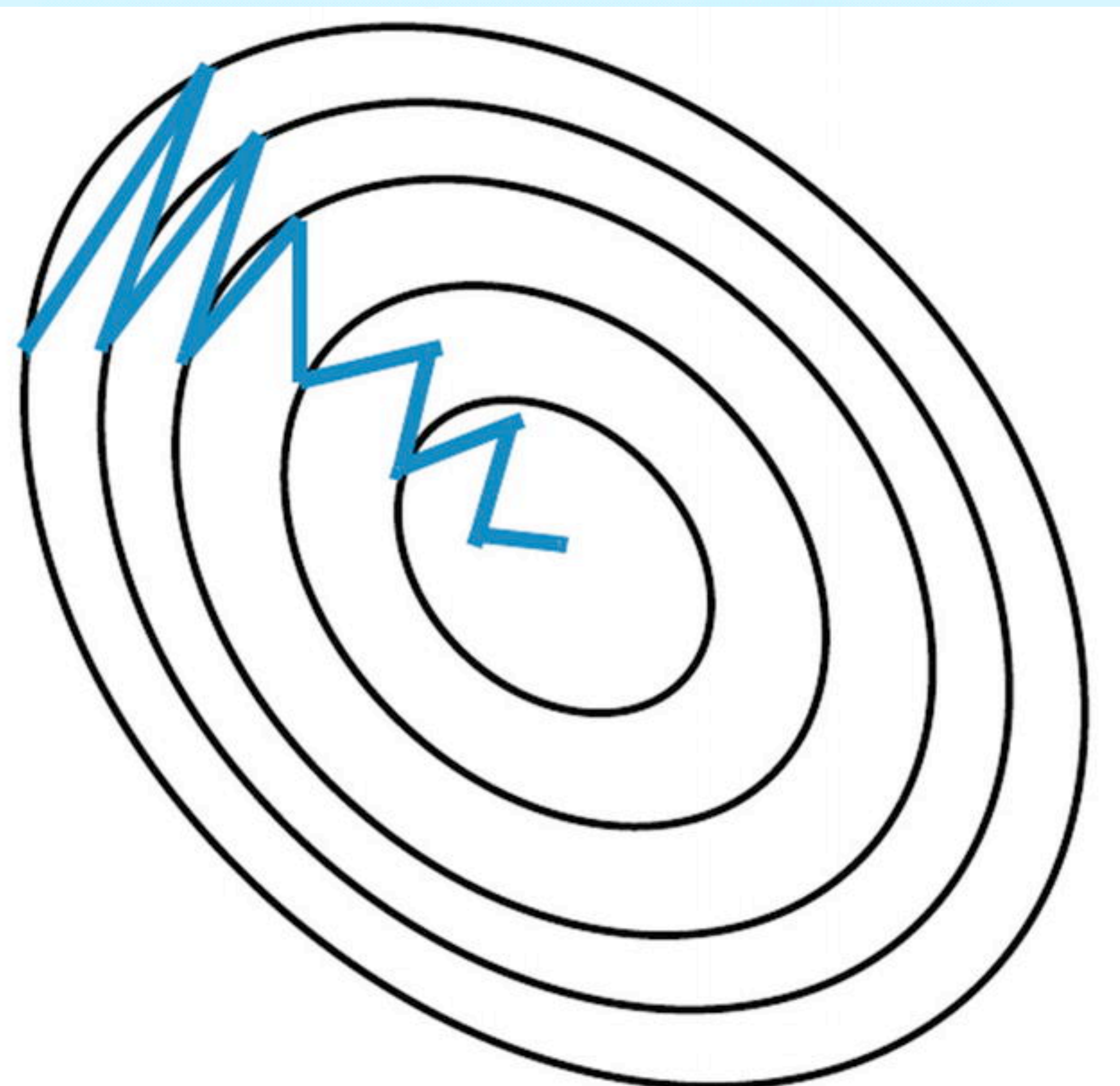
Bei uns ist das erstmal einfach der Faktor mit welchem der Gradient abgezogen wird:

```
def sgd_update(weights, learning_rate, gradient):
  """Update weights using a gradient and learning rate."""
  return [w - learning_rate * g for w, g in zip(weights, gradient)]
```

SGD with Momentum



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum